NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

TESTING AUTOMATION TOOLS FOR SECURE SOFTWARE DEVELOPMENT

by

Christopher Eatinger

June 2007

Thesis Advisor:	Mikhail Auguston
Second Reader:	Chris Eagle

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2007	Master's Thesis

4. TITLE AND SUBTITLE Testing Automation Tools for Secure Software Development	5. FUNDING NUMBERS
6. AUTHOR(S) Eatinger, Christopher J.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	

13. ABSTRACT (maximum 200 words)

Software testing is a crucial step in the development of any software system, large or small. Testing can reveal the presence of logic errors and other flaws in the code that could cripple the system's effectiveness. Many flaws common in software today can also be exploited to breach the security of the system on which the software is running. These flaws can be subtle and difficult to find. Frequently it takes a combination of multiple events to bring them out. Traditional testing techniques focus on dealing with errors as they arise during normal operation of the system. This technique is not particularly effective. Thus, recent research has focused on developing new, more effective software testing techniques. Two such techniques are combinatorial testing and fuzz testing.

This thesis explores the effectiveness of combining both combinatorial testing and fuzz testing into a single software testing tool to aid in the discovery of subtle system flaws. The tools developed for testing automation by this thesis will aid in the development of secure software, and bolster the ranks of testing techniques available to future developers.

14. SUBJECT TERMS Software Testing, Fuzzing, Combinatorial Testing	15. NUMBER OF PAGES 81
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

TESTING AUTOMATION TOOLS FOR SECURE SOFTWARE DEVELOPMENT

Christopher J. Eatinger
Civilian, Naval Postgraduate School
B.A., Oberlin College, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2007

Author: Christopher Eatinger

Approved by: Mikhail Auguston
Thesis Advisor

Chris Eagle
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software testing is a crucial step in the development of any software system, large or small. Testing can reveal the presence of logic errors and other flaws in the code that could cripple the system's effectiveness. Many flaws common in software today can also be exploited to breach the security of the system on which the software is running. These flaws can be subtle and difficult to find. Frequently it takes a combination of multiple events to bring them out. Traditional testing techniques focus on dealing with errors as they arise during normal operation of the system. This technique is not particularly effective. Thus, recent research has focused on developing new, more effective software testing techniques. Two such techniques are combinatorial testing and fuzz testing.

This thesis explores the effectiveness of combining both combinatorial testing and fuzz testing into a single software testing tool to aid in the discovery of subtle system flaws. The tools developed for testing automation by this thesis will aid in the development of secure software, and bolster the ranks of testing techniques available to future developers.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Dr. Mikhail Auguston and Professor Chris Eagle for providing the inspiration for this thesis, and also for their support and encouragement throughout the whole process. I would also like to thank Tim Levin for sharing his knowledge about the TCX project and for his input during the initial stages of this thesis.

This material is based on work supported by the National Science Foundation under Grant No. DUE—0414102 and by the Office of Naval Research. I would like to thank the National Science Foundation and the Office of Naval Research for their contributions. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or of the Office of Naval Research.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Section I consists of an introduction to the basic ideas behind software testing and the purpose of this thesis as well as a brief introduction to some of the background issues motivating this study.

Section II provides a more thorough exploration of the backgrounds of both combinatorial testing and randomized testing. The testing requirements of Common Criteria EAL6 are presented as well as a brief summary of the TCX project.

Section III explores the front-end input specification language and the back-end test profiles. The BNF grammars for both are presented as are the various metavariable constructs of the test profiles.

Section IV describes the implementation and architecture of the tool developed by this thesis. The internal data structures of the tool are outlined and explored. The interface to the Combinatorial Test Services library is described as is the architecture of the chooser, the randomized testing component of this tool.

Section V presents the experiments and evaluation that was performed on the tool. The input parameters are provided and the outputs are shown.

Section VI finishes off with a brief listing of the conclusions of this thesis and some recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

In recent years software security has received the spotlight of media attention. This surge of interest closely follows an explosion in the number of incidents reported each year. In 2002 the Computer Emergency Response Team (CERT) received 82,094 incident reports. The following year they received 137,529 incident reports. After 2003 attacks against internet-connected systems became so commonplace that CERT ceased publishing the number of incidents reported [1].

In 2006 CERT released the "eCrime Watch Survey" which presented the results of a poll conducted by CSO magazine whose readers consist predominantly of Chief Security Officers and other security and law enforcement professionals. Of the polls 434 respondents 72% reported that their organization was attacked by a virus, worm, or other malicious code in the preceding 12 months [9]. Most attacks of this nature take advantage of flaws present in the code to breach a system's security. When these flaws are discovered, often by uncovering viruses and other malicious code that takes advantage of them, they can be patched. However, patching a single flaw has the potential to introduce new flaws into the code.

This penetrate-and-patch approach to security though woefully bad at producing secure software is also, regrettably, a very common way of trying to achieve software security. A much better approach involves testing a system throughout its development. Each system module should be tested independently in addition to testing the system as a whole. Security is not an add-on feature. It must be built into a system from the earliest stages of its design and rigorously tested.

A. SOFTWARE TESTING

Testing is an essential part of any development project. It is the process whereby a system's security, correctness, and completeness are verified. Testing is intended to provide a reasonably measure of confidence that a system will operate as designed in the context in which it is deployed. For a system to operate as designed the functionality it

provides must be both necessary and complete. In other words, the system should accomplish that which it was designed to do and nothing more.

Occasionally, unintended functionality is discovered in a system. The developer response to this occurrence is often the jovial remark, “It’s not a bug. It’s a feature.” However, this response comes from the outdated mindset that each and every bit of functionality your system provides increases its value. In fact the opposite is true. Unintended functionality can not have been tested during a system’s development because the developers didn’t know it was there. If part of the system was left untested then there can be no confidence in the security, correctness, and completeness of the system as a whole. Without that confidence a system loses much of its worth.

The role of software testing is not only to assure that a system does everything it is supposed to do, but also that a system does nothing more then what it is designed to do. This second aspect of software testing is often neglected in favor of the first because it frequently takes a good deal of time to provide a reasonably measure of confidence that a system does nothing more then what it is designed to do.

B. PURPOSE OF STUDY

The immediate purpose of this research is to build a software testing tool that can help to provide assurances that a system both meets its design requirements and contains no excess functionality. Some techniques for providing such assurances have been developed in the past, but no single technique has provided a complete solution to the problem. Thus, this thesis will explore the effectiveness of combining two such techniques into a single tool.

The more general purpose of this research is to improve the available methods of software testing. Currently, there are a number of fads in software development each with their own buzz words like “extreme” and “agile”. Each of these fads comes with its own testing methodology. However, the majority of them focus on assuring that a system does everything it is supposed to do, that it is complete. There are few available tools that can assure a system’s lack of excess functionality. The goal of this research is to provide a tool that accomplishes both.

II. BACKGROUND AND REVIEW OF RELATED WORK

This section describes the two testing techniques that will form the basis for the tools developed within this thesis. A brief history each technique is provided along with an analysis of its strengths and weaknesses. The testing requirements for software evaluated at Common Criteria EAL6 are listed as is a brief synopsis of the Trusted Computing Exemplar (TCX) Project.

A. RANDOMIZED TESTING

This form of testing, referred to often as fuzz testing, got its start in the late 1980s. The first paper on randomized testing was titled *An Empirical Study of the Reliability of UNIX Utilities* and was published by Barton Miller et al. in 1990. The impetus for this paper came, as the author puts it, "on a dark and stormy night."

> One of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. It was a race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that the basic utilities should not crash… on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us [to] believe that there might be serious bugs lurking in the systems that we regularly used [2].

Miller then goes on to explain how he and his co-authors built a program to generate random characters which could then be passed to any of the 90 different utility programs that they tested. A program was considered to fail this test if it crashed or hung after being fed a string of random characters. It should be noted that many of the utilities that failed this test underwent commercial product testing. Miller emphasized that this method of testing "is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability [2]."

Exhaustive testing is what every developer strives for when testing their systems. However, it is often the case that exhaustive testing would be far too costly either in time, processor cycles, or money. Thus, most testing methodologies strive to approximate exhaustive testing as best they can, and of all such attempts fuzz testing may seem to be one of the more naive methods. However, Miller addresses this issue,

> While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs in impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states [2].

The ability of random testing to discover fatal program bugs has not diminished over the years. There have been three subsequent papers by Miller et al. Recall that the original paper tested only command line utilities on a number of different flavors of UNIX and found that an average of 25-33% of the programs tested failed the test. The 1995 “Fuzz Revisited” Report [3] again tested UNIX command line utilities on various unixes, but in addition to the command line utilities several X-Windows applications were also tested. This time over 40% of the command line programs and 25% of the X-Windows applications failed the test. The 2000 Windows NT Fuzz Report [4] tested over 30 GUI-based applications on Windows NT by sending streams of random keyboard and mouse events and streams of random Win32 messages. An unfortunate 46% of the applications tested crashed or hung when subjected to the random stream of keyboard and mouse input, and an astounding 100% of the applications crashed or hung when subjected to a random stream of Win32 messages. The most recent 2006 Mac OSX Fuzz Report [5] tested both the command line utilities and a number of the GUI-based utilities that ship with Mac OSX. Of the 135 command line utilities tested 7% or 10 of them crashed and none hung. Of the thirty GUI-based applications tested 73% or 22 of them crashed or hung.

Despite the fact that randomized testing has been around for over 15 years it continues to be effective at finding software flaws. Indeed, it’s effectiveness at finding flaws in GUI applications appears to have increased over that time period. It would appear that systems are still being developed with a features-over-reliability mentality.

B. COMBINATORIAL TESTING

This form of testing comes from the combination of certain software design methodologies and a mathematical construct. One of the first papers on the topic was *The Combinatorial Design Approach to Automatic Test Generation* published by David Cohen et al. in 1996. In it he describes the motivation for combinatorial testing as follows,

> Designing a system test plan is difficult and expensive. It can easily take several months of hard work. A moderate-size system with 100,000 lines of code can have an astronomical number of possible test scenarios. Testers need a methodology for choosing among them. The ISO 9000 process gives some guidance, specifying that each requirement in the requirements document must be tested. However, testing individual requirements does not guarantee that they will work together to deliver the desired functionality.
>
> The combinatorial design method… can reduce the number of tests needed to check the interworking of system functions. Combinatorial designs are mathematical constructions widely used in medical and industrial research to construct efficient statistical experiments [6].

Any sufficiently complicated system will have far too many possible combinations of inputs to test them all. Similar to randomized testing, combinatorial testing attempts to help the developer choose which subset of possible input combinations to test. While randomized testing advocated a completely random combination of inputs for each test scenario combinatorial testing advocates a somewhat more structured approach.

> To design a test plan, a tester identifies parameters that determine possible scenarios for the system under test (SUT). Examples of such test parameters are SUT configuration parameters, internal SUT events, user inputs, and other external events. For example, in testing the user interface software for a screen-based application, the test parameters are the fields on the screen. Each different combination of test parameter values gives a different test scenario. Since there are often too many parameter combinations to test all possible scenarios, the tester must use some methodology for selecting a few combinations to test.

> In the combinatorial design approach, the tester generates tests that cover all pairwise, triple, or n-way combinations of test parameters specified in formal test requirements. Covering all pairwise combinations means that for any two parameters p1 and p2 and any valid values v1 for p1 and v2 for p2, there is a test in which p1 has the value v1 and p2 has the value v2 [6].

As the cardinality of the n-way combinations is varied there is a tradeoff between the level of coverage and the number of test scenarios required. As the cardinality decreases so too does the number of test scenarios generated and the quality of the code coverage. In the Cohen paper testers typically relied on either pairwise or triple coverage.

> An empirical study of user interface software… found that most field faults were caused by either incorrect single values or by an interaction of pairs of values. Our code coverage study also indicated that pairwise coverage is sufficient for good code coverage. The seeming effectiveness of test sets with a low order of coverage such as pairwise or triple is a major motivation for the combinatorial design approach.

Since Cohen's paper more and more research has been conducted exploring the effectiveness of combinatorial testing in general and pairwise testing specifically. Several papers have come out of IBM's Haifa Research Laboratory on the subject; among them *Software and Hardware Testing Using Combinatorial Covering Suites* and *Problems and Algorithms for Covering Arrays* both published by Alan Hartman in 2003 and 2002 respectively [7] [8]. In addition to these papers IBM has also released a library called Combinatorial Test Services.

1. IBM Combinatorial Test Services Tool

This tool serves as one part of a system test plan based on combinatorial testing. Its basic roll is to take a list of possible input values for the system under test and generate a list of tuples representing all n-way combinations of those input values.

> The Combinatorial Test Services (CTS) is a software library for generation and manipulation of testing input data or configurations. CTS enables the user to generate small test suites with strong coverage properties, choose regression suites, and perform other useful operations for the creation of systematic software test plans…

As an example, consider the testing of an Internet site that must function correctly on three operating systems (Windows®, Linux®, and Solaris), two browsers (Explorer and Netscape), three printers (Epson, HP, and IBM), and two communication protocols (Token Ring and Ethernet). Although there are 36 (=3X2X3X2) possible test configurations, the nine tests in Figure 1 cover all the interactions between different pairs of parameters of the system.

The interactions between operating systems and printers are all covered precisely once, but some interactions between operating systems and browsers are covered more than once. For example, Windows and Explorer are tested together twice in the test suite [10].

Operating System	Browser	Printer	Protocol
Windows	Explorer	Epson	Token Ring
Windows	Netscape	HP	Ethernet
Windows	Explorer	IBM	Ethernet
Linux	Netscape	Epson	Token Ring
Linux	Explorer	HP	Ethernet
Linux	Netscape	IBM	Token Ring
Solaris	Explorer	Epson	Ethernet
Solaris	Netscape	HP	Token Ring
Solaris	Explorer	IBM	Ethernet

Table 1. CTS Example [10]

C. COMMON CRITERIA EAL6 TESTING REQUIREMENTS

The goal of this thesis is to facilitate the development of secure software that meets or exceeds the standards such as those put forth by the Common Criteria standard. When a product is evaluated under the Common Criteria standard it is assigned an Evaluation Assurance Level (EAL) which reflects the assurance requirements that were fulfilled during the evaluation. For example, Windows 2000 with Service Pack 3 was evaluated at EAL4+ indicating that it exceeded the assurance requirements of EAL4 but did not meet the requirements of EAL5. The reader may draw their own conclusions about the quality of EAL4+ software. However, this thesis strives to aid in the

development of software that can be evaluated at EAL6. As such, a short description of the testing requirements for EAL6 follows.

> The class "Tests" encompasses four families: Coverage (ATE_COV), Depth (ATE_DPT), Independent testing (ATE_IND) (i.e. functional testing performed by evaluators), and Functional tests (ATE_FUN). Testing provides assurance that the [Target of Evaluation (TOE) Security Functionality (TSF)] behaves as described (in the functional specification, TOE design, and implementation representation)…
>
> [The objective of the coverage] family establishes that the TSF has been tested against its functional specification. This is achieved through an examination of developer evidence of correspondence…
>
> The components in [the depth] family deal with the level of detail to which the TSF is tested by the developer. Testing of the TSF is based upon increasing depth of information derived from additional design representations and descriptions (TOE design, implementation representation, and security architecture description).
>
> The objective is to counter the risk of missing an error in the development of the TOE. Testing that exercises specific internal interfaces can provide assurance not only that the TSF exhibits the desired external security behaviour, but also that this behaviour stems from correctly operating internal functionality…
>
> Functional testing performed by the developer provides assurance that the tests in the test documentation are performed and documented correctly. The correspondence of these tests to the design descriptions of the TSF is achieved through the Coverage (ATE_COV) and Depth (ATE_DPT) families.
>
> This family contributes to providing assurance that the likelihood of undiscovered flaws is relatively small.
>
> The families Coverage (ATE_COV), Depth (ATE_DPT) and Functional tests (ATE_FUN) are used in combination to define the evidence of testing to be supplied by a developer…
>
> The objectives of [the independent testing] family are built upon the assurances achieved in the ATE_FUN, ATE_COV, and ATE_DPT families by verifying the developer testing and performing additional tests by the evaluator [11].

For a more thorough explanation of the testing requirements of EAL6 please see the Common Criteria v3.1 Part 3, Section 15, Class ATE: Tests.

D. TCX PROJECT

The purpose of this project is to provide an example of how trusted computing systems and components can be constructed.

> The TCX project is constructing a separation kernel that will be high assurance and suitable for use in simple embedded systems. To guide the kernel development, we have created a reusable high assurance development framework. The main emphasis of this multifaceted research and development initiative is to transfer knowledge and techniques for high assurance trusted system development [to] new developers, evaluators and educators [12].

It is expected that the work and tools produced by this thesis will aid the TCX project to meet its high assurance goals.

THIS PAGE INTENTIONALLY LEFT BLANK

III. LANGUAGE SPECIFICATION AND TEST DELIVERY FRAMEWORK

The primary goal of this thesis is to make a testing tool that is as universal as possible. Software security is not a language dependant concept. There are a great many programming languages out there, and more are being written every year. Regardless of a developer's chosen language they should strive to write secure, concise code. The tools developed herein can help them achieve this goal.

A combination of fuzz testing, combinatorial testing, and a custom test delivery framework make up the foundation of this tool. For the front-end, a simple, yet generic, input specification language allows the tester to describe each input parameter of the system and the classes of inputs it accepts. This input specification is processed by IBM's Combinatorial Test Services tool which generates a list of tuples that represents all n-way combinations[1] of input classes. The input specification is also given to a chooser class that will generate random values from a specified input class when queried. For the back-end, the tester writes a test profile which is used as a template for the generation of the test driver(s). Finally, the list of tuples, the machine parsed version of the profile, and a reference to the chooser are handed to a driver generator which uses all three to generate the test driver(s).

It is this combination of fuzz testing in the chooser, combinatorial testing in the IBM library, and an automated test delivery framework in the back-end that gives this tool the ability to rapidly uncover subtle system flaws. A pictorial version of this process can be seen in Figure 1.

[1] By default n is set to 2.

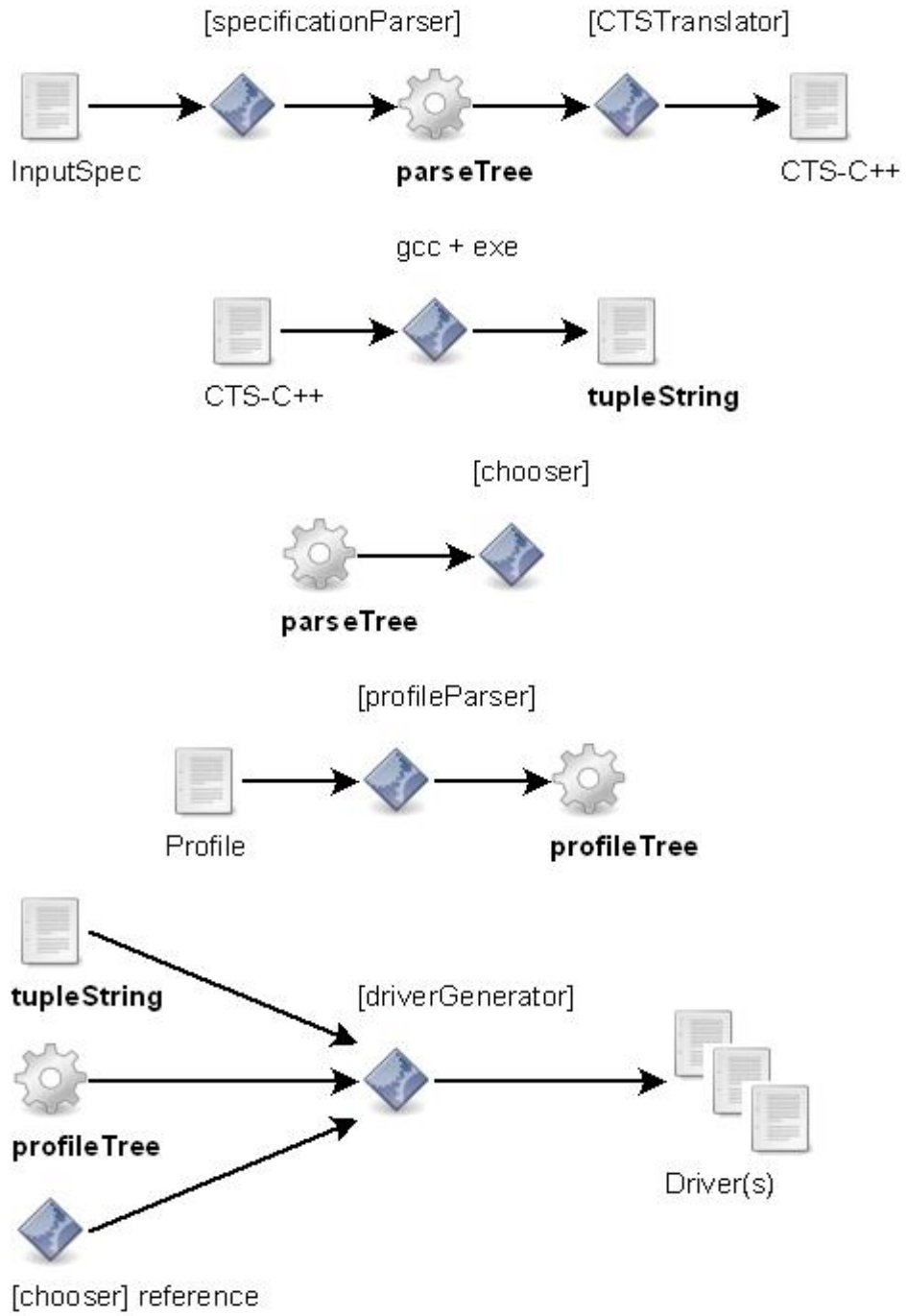

Figure 1. Program Flow (icons courtesy tango.freedesktop.org)

A. INPUT SPECIFICATION LANGUAGE

The input specification language for the front-end allows the tester to describe each input to the system being tested in a concise straightforward manner. Each input

description consists of a type and a series of input classes. Each class consists of a single value, a list of values, or a range of values.

1. Input Specification Language Grammar

```
<interface_description> ::= <target_description> [multiplicity]
                              (<input_description> ";")+
<target_description>    ::= "TARGET" [path]<filename>
<multiplicity>          ::= ("MULT" | "MULTIPLICITY") = (1|2|3|4)
<input_description>     ::= "int" <int_class>+ | "char" <char_class>+ |
                              "float" <float_class>+ |
                              "double" <double_class>+ |
                              "string" <string_class>+ |
                              "bool" <bool_class>+
<int_class>             ::= <integer> | <integer_range> | <integer_list>
<integer_range>         ::= "[" <integer> ".." [<integer>] "]" |
                              "[" [<integer>] ".." <integer> "]"
<integer_list>          ::= "(" <integer>+ ")"
<char_class>            ::= <set>
<float_class>           ::= <float> | <float_range> | <float_list>
<float_range>           ::= "[" <float> ".." [<float>] "]" |
                              "[" [<float>] ".." <float> "]"
<float_list>            ::= "(" <float>+ ")"
<double_class>          ::= <double> | <double_range> | <double_list>
<double_range>          ::= "[" <double> ".." [<double>] "]" |
                              "[" [<double>] ".." <double> "]"
<double_list>           ::= "(" <double>+ ")"
<string_class>          ::= <RE>
<bool_class>            ::= "random" | "true" | "false"
<integer>               ::= ["-"] <digit>+
<float>                 ::= ["-"] <digit>+ "." <digit>+ ["e" <integer>]
<double>                ::= ["-"] <digit>+ "." <digit>+ ["e" <integer>]

<RE>            ::= <union> | <simple-RE>
<union>         ::= <RE> "|" <simple-RE>
<simple-RE>     ::= <concatenation> | <basic-RE>
<concatenation>   ::= <simple-RE> <basic-RE>
<basic-RE>      ::= <star> | <plus> | <elementary-RE>
<star>          ::= <elementary-RE> "*"
<plus>          ::= <elementary-RE> "+"
<elementary-RE>   ::= <group> | <any> | <eos> | <char> | <set>
<group>         ::= "(" <RE> ")"
<any>           ::= "."
<eos>           ::= "$"
<char>          ::= any non metacharacter | "\" metacharacter
<set>           ::= <positive-set> | <negative-set>
<positive-set>  ::= "[" <set-items> "]"
<negative-set>  ::= "[^" <set-items> "]"
<set-items>     ::= <set-item> | <set-item> <set-items>
<set-item>      ::= <range> | <char>
<range>         ::= <char> "-" <char>
```

Figure 2. Input Specification BNF Grammar

Note that in the regular expression section of the above grammar a ‘metacharacter’ refers to any character that has a special meaning in standard regular expressions (e.g. ‘*’, ‘.’, ‘(‘, etc.).

The expansion for the string class represents one possible subset of regular expressions. Because of time constraints support for the string class has only been minimally implemented, and should be the first goal of any future work on this project.

2. Input Specification Examples

Here we will explore a basic example of an input specification. What follows is the specification for a system that has three inputs, each of which takes an integer.

```
TARGET ./test.profile MULT=2
int [1..10];
int [-5..5] 7;
int [20..100] -1000 (300 100 600 700 500 200 0 -5 800 900 1000);
```

Figure 3. Input Specification Example.

The first line of the specification defines two parameters. The first is the file name for test profile that is associated with this specification. The second, optional part of the first line allows the tester to specify the cardinality of the n-way combinations that will be generated by the Combinatorial Test Services tool. If this later part is left out, the system defaults to pairwise combinations. The next three lines each represent an input description. The type of each description is specified first followed by one or more input classes for that description. For the first description there is only one input class: a range from one to ten. The second description consists of two input classes: a range from negative five to positive five and the singleton value seven. The final description contains one of each type of input class: a range, a singleton, and a list of ten values.

B. TEST PROFILES

The test profiles utilized by the back-end use a combination of system specific code and metavariable constructs. These constructs come in one of three varieties: simple

metavariables, source group repetition constructs, and intratuple repetition constructs. If a profile does not contain any source group repetition constructs then a test driver will be generated for each tuple in the list of tuples output by the Combinatorial Test Services tool. If, on the other hand, a profile contains a source group repetition construct then only one test driver will be generated.

1. Test Profile Grammar

```
<profile>                  ::= [“METAVARIABLESYMBOL=” <char>+]
      (constant_string | <metavariable> |
      <source_group_repetition> |
      <intratuple_repetition>)*
<metavariable>             ::= <METAVARIABLESYMBOL> natural_number
<source_group_repetition> ::= <METAVARIABLESYMBOL> "{"
      (constant_string | <metavariable> | <intratuple_repetition>)*
      <METAVARIABLESYMBOL> "}"
<intratuple_repetition>   ::= <METAVARIABLESYMBOL>
      "[" natural_number [".." <natrual_number>] "]"
      "{"(constant_string | <metavariable
      <intratuple_repetition>)* <METAVARIABLESYMBOL> "}"
```

Figure 4. Test Profile BNF Grammar

The grammar for the test profiles is purposefully simple to maximize its usefulness. When a profile is parsed the code surrounding the metavariable constructs is simply copied and pasted character for character into any test drivers. Because of this it doesn’t matter whether the surrounding code is C++, Java, Perl, or Haskell. As long as the code doesn’t contain the metavariable symbol everything will just work. In the event that a profile is being written for a language that does use the default metavariable symbol a new symbol can be defined on the first line of the profile.

2. Metavariables

The most basic of these constructs simply consists of the metavariable symbol[2] followed by a number, n. When this construct is encountered in a profile it is replaced by a random value from the n^{th} input class in the current tuple. If a second metavariable construct with the same n is encountered while still processing the same tuple the value

[2] By default this symbol is represented by two percent signs, ‘%%’.

inserted will be the same as the one used for the previous substitution. If a new value is desired the second time a similar metavariable is encountered then the n can be prefaced with a caret, '^'. This will result in a new random value from the n^{th} input class in the current tuple. The example in Figure 5. illustrates how a simple function can be tested and verified using the register-like behavior of the metavariables.

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;

  answer = addThree(%%1, %%2, %%3);

  if(answer == %%1 + %%2 + %%3) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", %%1, %%2,
%%3, answer);
  }

}
```

Figure 5. Sample Profile with Metavariables

3. Intratuple Repetition Constructs

These constructs consist of the metavariable symbol and either a single value or a range enclosed in brackets followed by an open brace, a block of code, the metavariable symbol again, and a closing brace. The block of code can be comprised of any combination of metavariables, other intratuple repetition constructs, and the language of the system being tested. If there is only a single value in the initial brackets then the enclosed block of code will be repeated that many times. If the initial brackets contain a range then the enclosed block will be repeated a random number of times between the range's minimum and maximum values. Any metavariables in the enclosed block will use the same tuple that was being processed when the intratuple construct was first

encountered. The admittedly contrived example in Figure 6. shows how the intratuple repetition construct can be used to reproduce a block of code some random number of times in each test driver, in this case the character ‘A’ from 1 to 10000 times. On the following line, this example shows how the intratuple repetition construct can be combined with metavariable repetition. In this case, the intratuple construct will be replaced by 65536 random values from the 3rd input class in the current tuple in each test driver.

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;

  answer = addThree(%%1, %%2, %%3);

  if(answer == %%1 + %%2 + %%3) {
    printf("Successful\n");
    printf("%%[1..128]{A%%}");
    printf("%%[256]{%%^3%%}");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", %%1, %%2,
%%3, answer);
  }

}
```

Figure 6. Sample Profile with Intratuple Repetition

Note that in the sample profile in the figure above the double percent metavariable symbol appears inside of a call to printf. Double percent is a valid value in this situation. If the double percent was desired as input to the printf call then an alternate metavariable symbol would need to be specified at the beginning of this profile.

4. Source Group Repetition Constructs

This construct consists of the metavariable symbol followed by an open brace, a block of code, the metavariable symbol again, and a closing brace. If a profile contains

the source group repetition construct then only a single test driver will be generated. In that test driver, the block of code enclosed by the source group repetition construct will be repeated once for each tuple in the tuple list. Any metavariables in the enclosed block will use the first tuple for the first repetition, the second tuple for the second repetition, and so on. The example in Figure 7. shows how the source group repetition construct can be used to compact the multiple drivers generated by the example in Figure 5. into a single test driver.

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;
%%{
  answer = addThree(%%1, %%2, %%3);

  if(answer == %%1 + %%2 + %%3) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", %%1, %%2,
%%3, answer);
  }
%%}

}
```

Figure 7. Sample Profile with Source Group Repetition

IV. SOFTWARE IMPLEMENTATION, ARCHITECTURE, AND DESIGN DECISIONS

A number of languages were considered for the development of a testing tool, among them Java, Ruby, and Perl. The choice was made to go with Ruby based on its strengths as both a scripting language and a fully featured object oriented language. The object oriented nature of Ruby and its programmer-friendly development style were both extremely helpful when developing the front and back-ends. Also, the ease with which Ruby does inter-application processing meant that combining the front and back-ends with the Combinatorial Test Services tool was almost trivial. Finally, Ruby's scripting language roots made for easy processing of both the front-end input specifications and the back-end profiles.

For an overview of the architecture please see Figure 1. . The testing tool developed in this thesis has three main phases of operation. For the tool to function properly the tester must have written both an input specification for the front-end and a test profile for the back-end. Both the input specification and the test profile should be stored in files accessible by the testing tool.

In phase one the input specification is read in and processed by the specificationParser class. The output of this processing is a tree structure which consists of a combination of hashes and arrays hereafter referred to simply as the parse tree. Next, this parse tree is given as input to the ctsTranslator class which builds a Combinatorial Test Services C++, hereafter CTS-C++, string that is output to a temporary file. Once the parse tree and the CTS-C++ file have been generated phase two can begin.

Phase two consists of three relatively independent processes. These processes can be run in any order, but all three must be completed before phase three can begin. For the sake of clarity we will use the order shown in Figure 1. .

- For the first process, the CTS-C++ file is compiled into an executable by a C++ compiler[3]. That executable is then run, and the output is captured by the testing tool. This output consists of a list of tuples hereafter referred to as the tuple string.
- In the second process, the parse tree is used to initialize an instance of the chooser class.
- In the third process, the test profile is read in and processed by the profileParser class. The output of this processing is another tree structure made up of a combination of hashes and arrays hereafter referred to as the profile tree.

Once these three processes are completed it is time to move on to the third and final phase, test driver generation.

To begin the final phase, the tuple string, the profile tree, and a reference to the chooser instantiated in phase two are all used to initialize an instance of the driverGenerator class. This driverGenerator uses the profile tree as a template for generating the final test drivers. If the profile tree contains a source group repetition construct then only a single test driver will be generated, otherwise the driverGenerator will output one test driver for each tuple in the tuple string. For each driver, a depth first algorithm is used on the profile tree. Leaves that are constant strings are copied directly into the final driver. For leaves that are metavariables the chooser is used to supply a random value from the appropriate input class in the current tuple. A recursive call is made when leaves that are either intratuple repetition constructs or source group repetition constructs are encountered.

[3] The compiler used for this thesis was the GNU C Compiler Suite.

A. INTERMEDIATE DATA STRUCTURES

1. Input Specification Parse Tree

The image in Figure 8. shows the layout of the parse tree that is generated when an input specification is parsed by the specificationParser class. The elements with squared ends represent arrays, those with pointed ends represent terminal values, and those with rounded ends represent hashes. If the text in a hash element is quoted then it represents the exact key that is used for that element. If the text is not quoted then it represents the class of keys used for that hash.

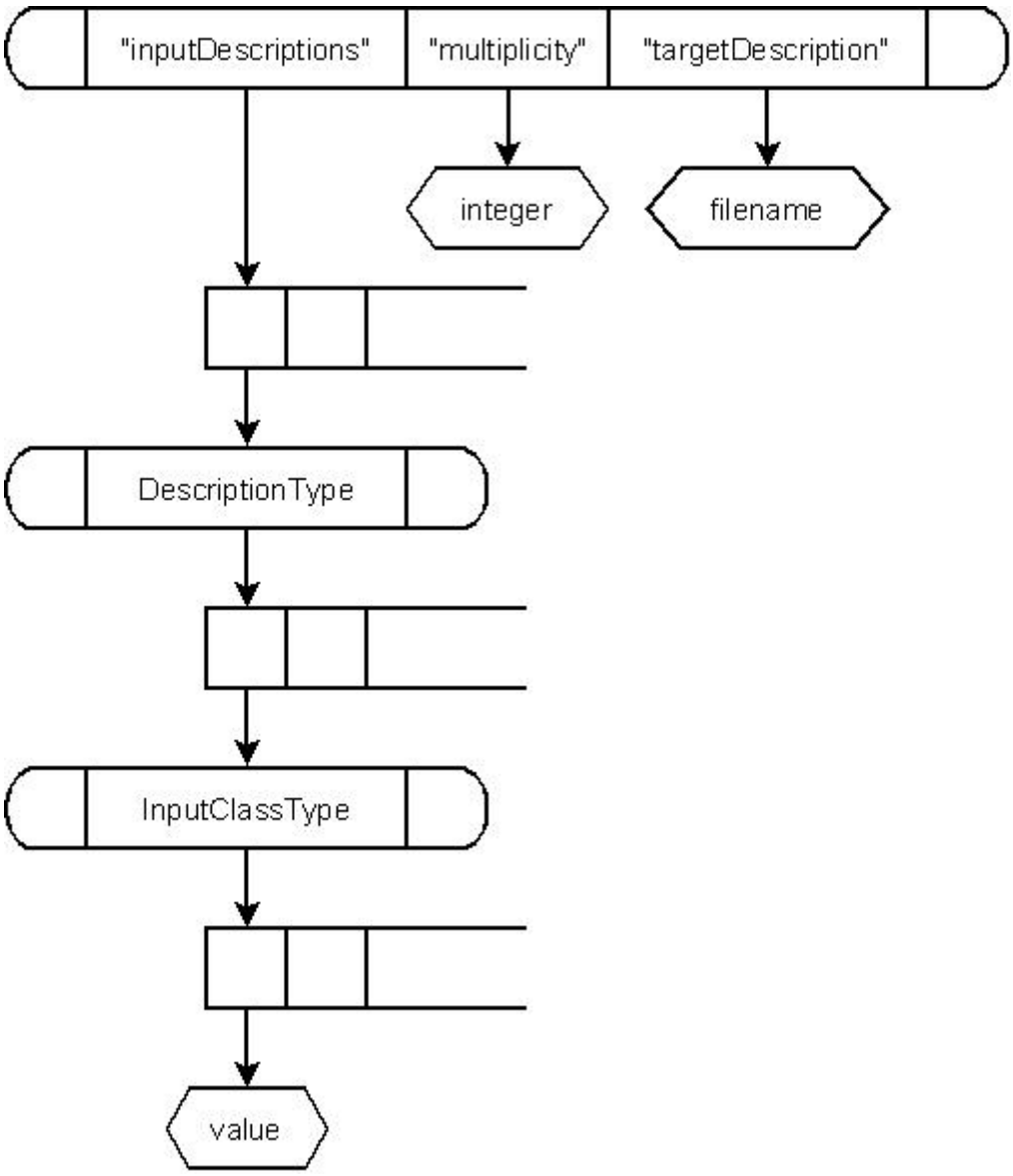

Figure 8. Input Specification Parse Tree Layout

The root element of the parse tree is a hash with three key/value pairs. The "multiplicity" key corresponds to an integer value representing the cardinality of the n-way combinations that will be generated by the Combinatorial Test Services tool. The "targetDescription" key corresponds to a string value that is the filename of the profile that will be used during driver generation. The "inputDescriptions" key corresponds to an array where each element represents a single input description. Each input description is represented by a hash with a single key/value pair. The key in this case is the type of the

input description, “int”, “char”, “double”, etc., and the value is an array where each element represents a single input class. Each input class is represented by another hash with a single key/value pair. In this case, the key is the type of input class, “singleton”, “range”, or “list”, and the value is an array where each element is an actual value of the type of the description.

2. Profile Tree

The image in Figure 9. shows the layout of the profile tree that is generated when a test profile is read in and processed by the profileParser class. The legend is the same as that used for the previous diagram with two additions. First, the makeup of the lower two, red arrays mirrors that of the first red array and as such are not included in this diagram. Second, a dotted outline around an element in a hash indicates that that element is optional.

The root element of the profile tree is a hash with two key/value pairs. The “repetition” key corresponds to a Boolean value which, when true, indicates the presence of a source group repetition construct in the profile tree. The “tree” key corresponds to an array whose elements are hashes with one of four different compositions.

The first possible hash has a single key/value pair and represents a constant string. The “string” key refers to a string value that is just that. The second possible hash also has a single key/value pair, but it represents a source group repetition construct. The “sourcegroup” key corresponds to an array whose makeup mirrors that of the “tree” array in the root level element. The third possible has two key/value pairs and represents a metavariable in the profile. The “metavariable” key corresponds to the integer value of the metavariable. The “new” key corresponds to a Boolean value which, when true, disables the register-like behavior of the metavariable. In other words, even if a similarly numbered metavariable was encountered earlier in the profile this metavariable will still be replaced by a new random value from the appropriate input class in the current tuple. The final possible hash can have either two or three key/value pairs, and it represents an intratuple repetition construct. The “low” and optional “high” keys refer to integer

values[4] which indicate the range from which the random number of repetitions will be drawn. The “intratuple” key corresponds to an array whose makeup mirrors that of the “tree” array in the root level element.

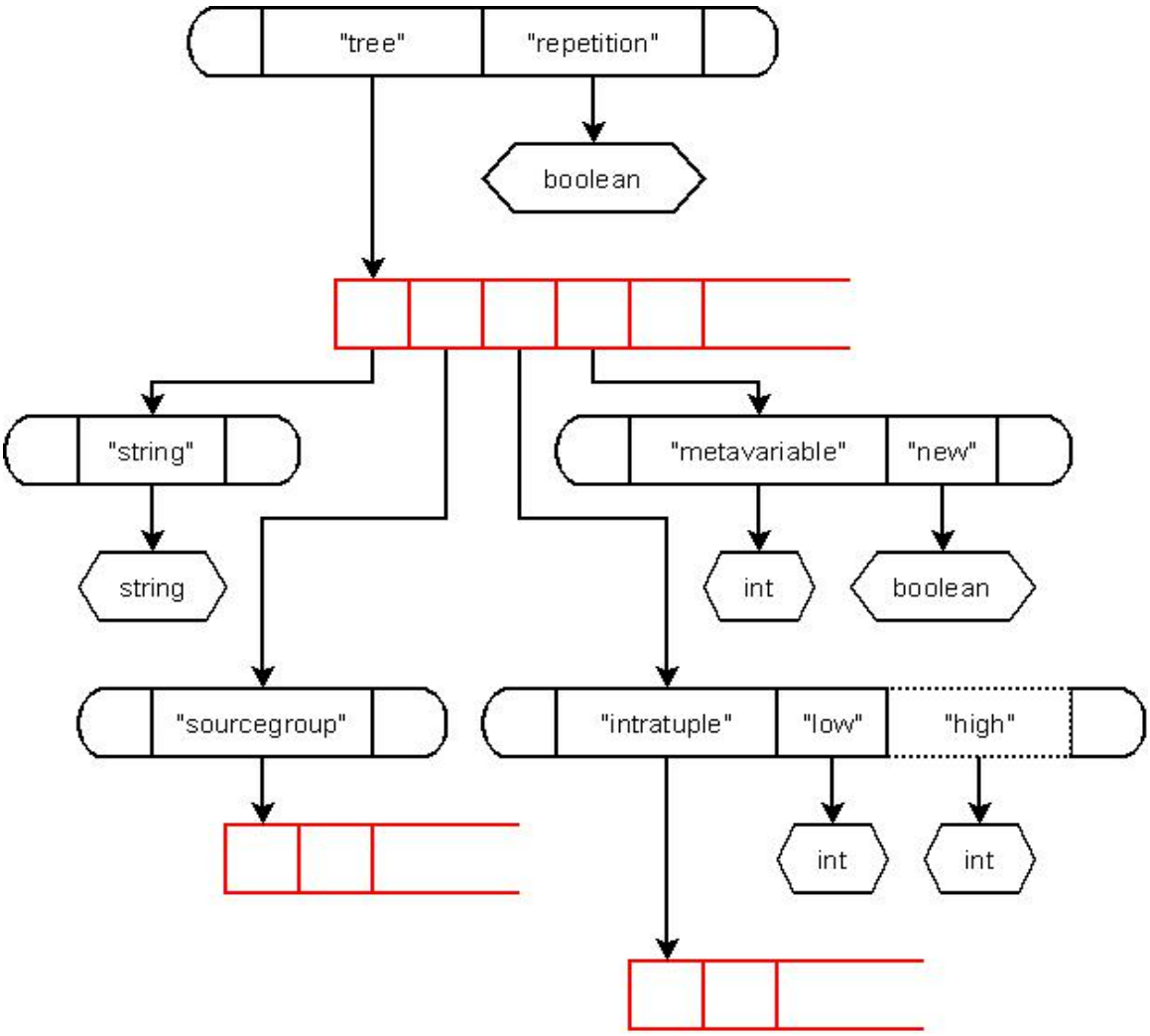

Figure 9. Test Profile Tree Layout

B. COMBINATORIAL TEST SERVICES USAGE AND INTERFACE

The Combinatorial Test Services library has a robust API that allows for the definition of multiple test suites using a variety of types and profiles. However, for the

[4] These values should be natural numbers.

purposes of the testing tool developed in this thesis it was not necessary to utilize all of the library's functionality. Instead of using the library's built-in string type to define each input class directly the decision was made to use the library's built-in integer type as an index to each input class. This made the ctsTranslator class much more straightforward.

The library uses a layered approach when defining a CTS-C++ file. On the lowest level, the library's built-in CTSInteger types are used to define indices for each input class. An instance of the library's CTSAttribute class is then defined to wrap each of these integers. Next, a CTSTestCaseProfile is defined and populated with the CTSAttributes. Finally, a CTSTestSuite is instantiated using this CTSTestCaseProfile which can then be used to build and print a test suite to get the final list of tuples. Each tuple in the final list is made up of indices which are used by the driverGenerator class in its queries to the chooser class for values from an input class.

C. PROFILE DESIGN PROBLEMS AND SOLUTIONS

A number of questions arose during the design process of the back-end test profiles. Should the profiles be built with a specific language in mind? Should nesting be allowed for the repetition constructs? What constructs should be built into the test profiles to begin with? Should stand-alone metavariables be allowed outside of source group repetition constructs? The goal of this thesis was to create a tool that emphasized generality and consistency, and that was reflected in the answers to the questions posed above.

To ensure that this tool could be used by as many developers as possible the decision was made not to gear the test profiles towards any specific language. The choice of '%%' as the default metavariable symbol was made with a similar goal in mind since '%%' is uncommon in many of the major programming languages. Also, since testing often involves repetitive processes source group and intratuple repetition constructs were included to keep profiles to a manageable size. However, the inclusion of these two constructs in the profiles introduced several new issues.

One such issue was whether repetition constructs should be allowed to be nested inside one another. Because there was no foreseeable situation in which nested source

group repetition would be useful source group repetition was disallowed. However, the nesting of intratuple repetition constructs could be seen to be useful in some situations, therefore, it was allowed.

Another issue arose out of the way in which source group repetition constructs affect driver generation. Since the block of code enclosed by a source group construct is repeated once for each tuple, and only a single test driver is generated, allowing metavariables outside this construct would be inconsistent. Thus, the decision was made to exclude this possibility.

D. CHOOSER ARCHITECTURE

The input specification language, the Combinatorial Test Services library, and the back-end test profiles all introduce structure to the testing process. The chooser is where the random element that was found so effective in fuzz testing is introduced into the process. In its current form the chooser has two methods which return values from a given input class, valueFrom and newValueFrom. Both take two integer arguments. The first specifies which input description to use, and the second specifies which input class within that description to draw the value from. The valueFrom method implements the register-like behavior described earlier, and the newValueFrom method simply returns a new random value from the specified input class each time it is called.

Depending on the type of a given input class the chooser uses one of three classes to represent it internally: discreteSet, infiniteSet, and stringSet. Currently only the discreteSet and the infiniteSet are implemented. It is in these classes that the random choice logic is implemented for each type. For integer, character, and Boolean input classes the discreteSet class is used. For ranges, an array of all possible values for the input class is generated, and when queried for a value it returns a random element from the array. For floats, and doubles the infiniteSet class is used. For ranges, a random value from the difference between the low-end and the high-end is calculated and added to the low-end. For strings the stringSet class will be used.

V. EVALUATION AND EXPERIMENTS

In order to verify that the all the features described in this thesis worked as described a number of experiments were derived. The first involves a basic test profile with simple metavariable substitution on a system with three integer inputs. All three types of input classes, singleton, range, and list, are represented in the input specification for this system. The second uses the same system, but the test profile has been modified to include source group repetition with metavariable substition. The third also uses the same system, but this time the test profile involves intratuple repetition combined with metavariable substitution.

The same basic driver was used for all of these experiments, and can be seen in Figure 10. . It takes two arguments, the input specification and the test profile in that order. By customizing the filenames of both the input specification and the test profile this driver should be suitable for use in most testing scenarios. It is important to note that, in its current form, the driver must be run from the parent directory of the tool's source directory. This is not necessarily a requirement, but changing the various paths in the tool's source code is a somewhat involved process, and hence will not be discussed here. Also, note that there are two lines in this driver that have been commented out. These lines typically clean up the temporary files that are generated when the driver runs. However, in order to show what those files look like they were not included in these experiments.

In order for the tool to function properly the following system requirements must be met. The system must have a working installation of the GNU C Compiler suite of tools. There must be a valid Ruby installation on the system at or above version 1.8.4. The Combinatorial Test Services library must be installed, and it must be accessible to the afore mentioned compiler[5].

[5] For the purposes of this test the CTS library was installed in the tool's source directory, and the appropriate arguments were added to the call to g++.

```
#!/usr/bin/ruby -I./src/

require 'fileutils'
require "specificationParser"
require "ctsTranslator"
require "chooser"
require "profileParser"
require "driverGenerator"

if ARGV.size == 2
  inputSpec = ""
  File.open(ARGV[0], "r") { |fd|
    fd.each_line { |line| inputSpec += line }
  }

  specTree = SpecificationParser.new.parse(inputSpec)
  ctsDriver = CTSTranslator.new.translate(specTree)
  File.open("ctsDriver.cc", "w") { |fd|
    fd << ctsDriver << "\n"
  }
  if system("g++ ctsDriver.cc -I ./src/CTS/include/
./src/CTS/bin/linux/cts.a -o ctspgrm.out")
    ctsOutput = `./ctspgrm.out`
    ###FileUtils.rm("ctsDriver.cc")
    ###FileUtils.rm("ctspgrm.out")
  end
  profileString = ""
  File.open(ARGV[1], "r") { |fd|
    fd.each_line { |line| profileString += line }
  }
  chooser = Chooser.new(specTree)
  profile = ProfileParser.new()
  profileTree = profile.parse(profileString)

  Dir.mkdir("./gen_drivers") rescue nil
  generator = DriverGenerator.new(ctsOutput, profileTree, chooser)
  generator.generateDrivers

  puts "Done."
else
  puts "Usage: tool_driver INPUTSPEC PROFILE"
end
```

Figure 10. Sample Test Driver

A. EXPERIMENT 1: METAVARIABLE SUBSTITUTION

For this experiment the input specification shown in Figure 3. and the test profile shown in Figure 5. were used. The test ran successfully and generated the expected

ctsDriver.cc and ctspgrm.out files as well as 6 test drivers. The ctsDriver.cc file as well as the output from the ctspgrm.out and one of the test drivers are displayed in the following three figures.

```
#include "CTS.h"
#include <limits.h>

using namespace CTS_HRL;

int main()
{
      CTSIntegerType inputType0(1);
      CTSIntegerType inputType1(2);
      CTSIntegerType inputType2(3);

      CTSAttribute attr0("input0", inputType0);
      CTSAttribute attr1("input1", inputType1);
      CTSAttribute attr2("input2", inputType2);

      CTSTestCaseProfile profile;
      profile.addAttribute(attr0);
      profile.addAttribute(attr1);
      profile.addAttribute(attr2);

      CTSTestSuite test(profile);
      test.build(2,INT_MAX,true);
      test.setPrintMode("CSV");
      test.print();
      return 0;
}
```

Figure 11. Experiment 1 ctsDriver.cc

```
input0 input1 input2

 0  0  0

 0  1  0

 0  0  1

 0  0  2

-1  1  1

-1  1  2
```

Figure 12. Experiment 1 ctspgrm.out output

Note that the last two tuples have –1 as their first element. This is how the Combinatorial Test Services library indicates that the value chosen for that element does not matter. In Ruby the element at position -1 of an array is simply the last element, so it all works out.

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;

  answer = addThree(10, -2, 51);

  if(answer == 10 + -2 + 51) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 10, -2, 51,
answer);
  }

}
```

Figure 13. Experiment 1 Sample Driver

B. EXPERIMENT 2: SOURCE GROUP REPETITION WITH METAVARIBALE SUBSTITUTION

For this experiment the same input specification shown in Figure 3. was used, but this time the test profile was the one shown in Figure 7. . The test ran successfully and generated the expected ctsDriver.cc and ctspgrm.out files as well as the single expected test driver. The ctsDriver.cc file and the output from the ctspgrm.out were the same as those generated in the previous experiment and can be seen in Figure 11. and Figure 12. . The test driver is displayed in Figure 14. .

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;

  answer = addThree(2, -5, 89);

  if(answer == 2 + -5 + 89) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 2, -5, 89,
answer);
  }

  answer = addThree(5, 7, 66);

  if(answer == 5 + 7 + 66) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 5, 7, 66,
answer);
  }

  answer = addThree(1, 4, -1000);

  if(answer == 1 + 4 + -1000) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 1, 4, -1000,
answer);
  }

  answer = addThree(8, 0, 800);

  if(answer == 8 + 0 + 800) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 8, 0, 800,
answer);
  }

  answer = addThree(7, 7, -1000);

  if(answer == 7 + 7 + -1000) {
    printf("Successful\n");
  }
  else {
```

```
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 7, 7, -1000,
answer);
  }

  answer = addThree(1, 7, 600);

  if(answer == 1 + 7 + 600) {
    printf("Successful\n");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 1, 7, 600,
answer);
  }


}
```

Figure 14. Experiment 2 Sample Driver

C. EXPERIMENT 3: INTRATUPLE REPETITION WITH METAVARIABLE SUBSTITIUTION

For this experiment the same input specification shown in Figure 3. was used, but this time the test profile was the one shown in Figure 6. . The test ran successfully and generated the expected ctsDriver.cc and ctspgrm.out files as well as six test drivers. The ctsDriver.cc file and the output from the ctspgrm.out were the same as those generated in experiment 1 and can be seen in Figure 11. and Figure 12. . One of the six test drivers is displayed in Figure 15. .

```
#include <stdio.h>
#include "addthree.c"

using namespace std;

int main() {

  int answer;

  answer = addThree(10, 7, 24);

  if(answer == 10 + 7 + 24) {
    printf("Successful\n");

printf("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");

printf("7558902849396783239362222967598661595820985969433097358195904166
523829631008430666394495026726368987467509067915565858722674591665288339
728100408649929720486957717160595836374245232749783783629754737757958082
782827505695408259888643346027509226337262313248293092977891416464415510
077442221498869267939553180238054429139212587813461328039749943262329918
661438170887168424875782872666353248493324474912897957643912489734836573
737462357874176337088378964308478516079217340694035208775262331819383256
3723897594879726349");
  }
  else {
    printf("Unsuccessful!\n Values: %d %d %d\t Answer: %d", 10, 7, 24,
answer);
  }

}
```

Figure 15. Experiment 3 Sample Driver

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

A. SUMMARY

The tool developed in this thesis combines the strengths of combinatorial testing and fuzz testing with an input specification language and testing profiles. The union of these disparate testing techniques allows the developer to reap the benefits of both combinatorial testing's good code coverage and fuzz testing's ability to uncover subtle system flaws. The front-end input specification and back-end test profiles dramatically speed up the testing process. The tool developed in this thesis may now be integrated into any test plan to improve the reliability and security of the system being tested. Specifically, when the TCX project has advanced to the testing phase this tool should prove useful during that process.

B. FUTURE WORK

Two areas of future work come immediately to mind. The first would be to extend the functionality of this tool's string input class. The definition of a string range is somewhat ambiguous and needs to be resolved either by disallowing it entirely or by coming up with a consistent definition. Also, the string input class could be customized for dealing with hexadecimal and octal values (i.e. 0x and 0 preceding a string).

Also, the work done in this thesis can be seen as an extension of the JUnit method of software testing. In the future it might be advantageous to merge the methods developed for this tool with those of JUnit.

The second focus of possible future work would be in test validation. Currently, this tool only generates the test drivers and leaves the validation of those tests up to the developer. It should be possible to automate the validation phase just as the test generation phase was automated in this thesis. Such an addition to this tool would make it even more powerful asset to a developer tasked with creating a system test plan.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SOURCE CODE

A. SPECIFICATIONPARSER.RB

```
# This class is responsible for producing the parse tree of the input
# specification that is used by the other parts of this tool. Its use is
simple,
# instantiate it and call the parse method with a valid input
specification
# string.
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

require 'strscan'

class SpecificationParser
  attr_reader :input, :tokenList, :parseTree

  def initialize
    @input
    @tokenList = []
    @parseTree = []
  end

  # The tokenize regexp matches, in order, white space, names,
  # numbers (integers and floats), brackets, parentheses, range markers,
  # equals signs, and semicolons.
  #  regexp = %r{ ^( \s+ | \
  #                  [a-zA-Z_\.\/\\][a-zA-Z0-9_\.\/\\]* | \
  #                  [-+]?\d*\.?\d+([eE][-+]?\d+)? | \
  #                  \[ | \] | \
  #                  \( | \) | \
  #                  \.\. | \= | \; ) }x
  #
  # As a temporary fix to range markers being confused for names the
range
  # marker '..' test has been moved above the name test and slightly
modified
  # to not hit when the '..' is followed by a '/' or '\'
  #
  def tokenize(string)
    @input = StringScanner.new(string)
    while(!@input.eos?)
      if @input.scan(/\s+/) != nil
        # matched white space
      elsif @input.check(/\.\.([^\/\\]|$)/) != nil
        # matched range marker
        @tokenList << @input.scan(/\.\./)
```

```
      elsif @input.scan(/[a-zA-Z_\.\/\\][a-zA-Z0-9_\.\/\\]*/) != nil
        # matched name
        @tokenList << @input.matched
      elsif @input.scan(/[-+]?\d*\.?\d+([eE][-+]?\d+)?/) != nil
        # matched number
        @tokenList << @input.matched
      elsif @input.scan(/\[/) != nil
        # matched open bracket
        @tokenList << @input.matched
      elsif @input.scan(/\]/) != nil
        # matched close bracket
        @tokenList << @input.matched
      elsif @input.scan(/\(/) != nil
        # matched open parentheses
        @tokenList << @input.matched
      elsif @input.scan(/\)/) != nil
        # matched close parentheses
        @tokenList << @input.matched
      elsif @input.scan(/\=/) != nil
        # matched equals sign
        @tokenList << @input.matched
      elsif @input.scan(/\;/) != nil
        # matched semicolon
        @tokenList << @input.matched
      else
        # lexical error
        raise "#{@input.pre_match} ERROR #{@input.rest}"
      end
    end
  end

  # Start the parsing ball rolling on 'string' and returns the root of
  # the parse tree in the form of a hash.
  #
  def parse(string)
    begin
      @tokenList = []
      tokenize(string)
    rescue RuntimeError => boom
      print "Lexical error: Unrecognized symbol at " + boom
      exit
    end

    begin
      @parseTree = {
        "targetDescription" => parseTargetDescription(),
        "multiplicity" => parseMultiplicity(),
        "inputDescriptions" => parseInputDescription()
      }
      return @parseTree
    rescue RuntimeError => boom
      print "Parse error: " + boom
      exit
    end
  end
```

```
  # Returns the path to the profile as a string
  #
  def parseTargetDescription
    if @tokenList.first =~ /^TARGET$/
      @tokenList.shift
      if @tokenList.first =~ /(.*[\/\\])(.*)$/
        return @tokenList.shift
      else
        raise "Malformed profile name in target description."
      end
    else
      raise "Interface description must start with 'TARGET
[path]<filename>'"
    end
  end

  # Returns the multiplicity number as a string
  #
  def parseMultiplicity
    if @tokenList.first =~ /(^MULT$|^MULTIPLICITY$)/
      @tokenList.shift
      if @tokenList.first =~ /=/
        @tokenList.shift
      else
        raise "Multiplicity statement expected '='."
      end
      if @tokenList.first =~ /^[1234]$/
        return @tokenList.shift
      else
        raise "Malformed mulitplicity number."
      end
    else
      # Multiplicity not provided, go with default
      return "2"
    end
  end

  # Returns an array of hashes of arrays of input classes. Each input
description
  # is represented by a hash in the top level array with a single
key/value pair.
  # The key represents the type of the input description (int, char,
string, etc.),
  # and the value is an array of hashes each representing a single input
class for
  # that description.
  #
  def parseInputDescription
    inputDescriptions = []
    while !@tokenList.empty?
      inputDescription = {}
      inputClasses = []
      case @tokenList.first
      when /^int$/
```

```
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~ /[-+]?[0-
9]+/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"int" => inputClasses}
      when /^char$/
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~
/^(.|\n)$/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"char" => inputClasses}
      when /^float$/
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~ /[-
+]?\d*\.?\d+([eE][-+]?\d+)?/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"float" => inputClasses}
      when /^double$/
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~ /[-
+]?\d*\.?\d+([eE][-+]?\d+)?/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"double" => inputClasses}
      when /^string$/
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~ /.+/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"string" => inputClasses}
      when /^bool$/
        @tokenList.shift
        begin
          inputClasses << parseInputClass { |testee| testee =~
/^(true|false|True|False)$/ }
        end until @tokenList.first =~ /^;$/
        @tokenList.shift
        inputDescription = {"bool" => inputClasses}
      else
        raise "Input description type not recognized"
      end
      inputDescriptions << inputDescription
    end
    return inputDescriptions
  end

  # Returns a hash with a single key/value pair where the key is either
```

```
  # "singleton", "range", or "list", and the value is an array with
either
  # 1, 2, or 1.. elements respectively
  #
  def parseInputClass(&test)
    if @tokenList.first =~ /^\($/
      return parseList(&test)
    elsif @tokenList.first =~ /^\[$/
      return parseRange(&test)
    else
      return parseSingleton(&test)
    end
  end

  # Returns a hash with a single key/value pair where the key is "range"
  # and the value is a two element array with the first element
representing
  # the low end of the range and the second element representing the
high end
  #
  def parseRange(&test)
    @tokenList.shift # off the open bracket
    lowend = ""
    highend = ""
    if @tokenList.first =~ /^\.\.$/ # then no low end supplied
      @tokenList.shift
      if yield @tokenList.first
        highend = @tokenList.shift
        if @tokenList.first =~ /^\]$/
          @tokenList.shift
          return { "range" => ["infinity", highend] }
        else
          raise "Range terminator missing"
        end
      else
        raise "Range end element not of correct type:
#{@tokenList.first}"
      end
    elsif yield @tokenList.first
      lowend = @tokenList.shift
      if @tokenList.first =~ /^\.\.$/
        @tokenList.shift
        if @tokenList.first =~ /^\]$/ # then no high end supplied
          @tokenList.shift
          return { "range" => [lowend, "infinity"] }
        elsif yield @tokenList.first
          highend = @tokenList.shift
          if @tokenList.first =~ /^\]$/
            @tokenList.shift
            return { "range" => [lowend, highend] }
          else
            raise "Range terminator missing"
          end
        else
```

```
          raise "Range end element not of correct type:
#{@tokenList.first}"
        end
      else
        raise "Missing range marker, '..'"
      end
    else
      raise "Range begin element not of correct type:
#{@tokenList.first}"
    end
  end

  # Returns a hash with a single key/value pair where the key is "list"
  # and the value is an n-element array where n is the number of
elements
  # in the list supplied in the input specification
  #
  def parseList(&test)
    @tokenList.shift # off the open paren
    listArray = []
    while @tokenList.first !~ /^\)$/
      if yield @tokenList.first
        listArray << @tokenList.shift
      else
        raise "List element not of correct type: #{@tokenList.first}"
      end
    end
    @tokenList.shift # off the close paren
    return { "list" => listArray }
  end

  # Returns a hash with a single key/value pair where the key is
"singleton"
  # and the value is a single element array the one element of which is
the
  # single value supplied in the input specification
  #
  def parseSingleton(&test)
    if yield @tokenList.first
      return { "singleton" => [@tokenList.shift] }
    else
      raise "Singleton element not of correct type: #{@tokenList.first}"
    end
  end
end
```

B. CTSTRANSLATOR.RB

```
# This class translates a valid parse tree from the specificationParser
class
# into CTS-C++. To use it simply instatiate the class and call the
translate
# method with a parse tree.
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

class CTSTranslator
  attr_reader :input, :output

  def initialize
    @input = {}
    @output = ""
  end

  # Takes in a parse tree of the form generated by specificationParser
and
  # outputs C++ source code for the IBM CTS tool as a string.
  #
  def translate(parseTree)
    @input = parseTree
    inputDescriptions = parseTree["inputDescriptions"]
    tabLevel = 0

    # First we setup the CTS file header.
    @output << "\t"*tabLevel + "#include \"CTS.h\"" + "\n"
    @output << "\t"*tabLevel + "#include <limits.h>" + "\n\n"
    @output << "\t"*tabLevel + "using namespace CTS_HRL;" + "\n\n"
    @output << "\t"*tabLevel + "int main()" + "\n"
    @output << "\t"*tabLevel + "{" + "\n"
    tabLevel += 1

    # Setup CTS types for each input. For simplicities sake we will use
the
    # CTSIntegerType for all inputs the values of which will represent
indexes
    # into the parse tree branch for each respective input. This allows
us to avoid
    # converting our parse tree representations back into coherent
strings for CTS
    # to deal with and then having to remap the output from CTS back
onto our parse
    # tree.
    for i in (0...inputDescriptions.size)
      @output << "\t"*tabLevel + "CTSIntegerType
inputType#{i}(#{inputDescriptions[i].values[0].size});" + "\n"
    end
```

```
    @output << "\t"*tabLevel + "\n"

    # Here we wrap the above declared types in CTSAttributes.
    for i in (0...inputDescriptions.size)
      @output << "\t"*tabLevel + "CTSAttribute attr#{i}(\"input#{i}\",
inputType#{i});" + "\n"
    end
    @output << "\t"*tabLevel + "\n"

    # And here we declare the CTSTestCaseProfile and populate it with
the above
    # declared CTSAttributes.
    @output << "\t"*tabLevel + "CTSTestCaseProfile profile;" + "\n"
    for i in (0...inputDescriptions.size)
      @output << "\t"*tabLevel + "profile.addAttribute(attr#{i});" +
"\n"
    end
    @output << "\t"*tabLevel + "\n"

    # Lastly we declare the CTSTestSuite, give it the above declared
CTSTestCaseProflie,
    # tell it to build a test suite with the multiplicity provided in
the parse tree,
    # and print the resulting test suite. Print options are "ATS",
"CSV", and "TXT"
    # however, the setPrintMode function does not appear to have the
desired effect.
    # As such, we are currently limited to the default print mode of
"TXT".
    @output << "\t"*tabLevel + "CTSTestSuite test(profile);" + "\n"
    @output << "\t"*tabLevel +
"test.build(#{parseTree["multiplicity"]},INT_MAX,true);" + "\n"
    @output << "\t"*tabLevel + "test.setPrintMode(\"CSV\");" + "\n"
    @output << "\t"*tabLevel + "test.print();" + "\n"

    @output << "\t"*tabLevel + "return 0;" + "\n"
    tabLevel -= 1
    @output << "\t"*tabLevel + "}" + "\n"
    return @output
  end

end
```

C. CHOOSER.RB

```
# This class provides the architecture for the random choice logic for
this
# tool. To use it one must call the populate method on an instantiation
of this
# class with a parseTree generated by the specificationParser class.
After that
# it is simply a matter of using the valueFrom and newValueFrom methods
each of
# which takes two index values. The first represents the index into the
array of
# descriptions and the second represents the index into the array of
input
# classes in that description.
#
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

require "discreteSet.rb"
require "infiniteSet.rb"
require "stringSet.rb"

class Chooser
  attr_reader :descriptions

  def initialize(parseTree)
    @descriptions = []
    populate(parseTree)
  end

  # The populate method takes in a parse tree generated by
specificationParser
  # and populates an array of arrays of objects. Those objects
  #
  def populate(parseTree)
    @descriptions = []
    inputDescriptions = parseTree["inputDescriptions"]
    inputDescriptions.each { |description|
      classes = []
      case description.keys[0]
      when /int/
        description.values[0].each { |inputClass|
          values = []
          case inputClass.keys[0]
          when /range/
            values =
((inputClass.values[0][0].to_i)..(inputClass.values[0][1].to_i)).entries
          when /list/
            inputClass.values[0].each { |value|
```

```
              values << value.to_i
            }
          when /singleton/
            values = [inputClass.values[0][0].to_i]
          else
            raise "Invalid input class"
          end
          classes << DiscreteSet.new(values)
        }
      when /char/
        case inputClass.keys[0]
        when /range/
        when /list/
        when /singleton/
        else
          raise "Invalid input class"
        end
      when /float/
        case inputClass.keys[0]
        when /range/
        when /list/
        when /singleton/
        else
          raise "Invalid input class"
        end
      when /double/
        case inputClass.keys[0]
        when /range/
        when /list/
        when /singleton/
        else
          raise "Invalid input class"
        end
      when /string/
        case inputClass.keys[0]
        when /range/
        when /list/
        when /singleton/
        else
          raise "Invalid input class"
        end
      when /bool/
        case inputClass.keys[0]
        when /range/
        when /list/
        when /singleton/
        else
          raise "Invalid input class"
        end
      else
        raise "Input description type not recognized"
      end
      @descriptions << classes
    }
  end
```

```
  # Returns a value from the specified input description and class. If
  # a value has already been requested for a given input description
  # and class then that initial value will be returned for each
successive
  # call for a value from that input description and class.
  #
  def valueFrom(descriptionIndex, classIndex)
    @descriptions[descriptionIndex][classIndex].chooseValue
  end

  # Similar to the above method except where random choice is involved
  # a new choice is made.
  #
  def newValueFrom(descriptionIndex, classIndex)
    @descriptions[descriptionIndex][classIndex].chooseNewValue
  end

  # Resets the registers for each set in each description.
  #
  def reset
    @descriptions.each { |description|
      description.each { |set|
        set.reset
      }
    }
  end

end
```

D. PROFILEPARSER.RB

```
# This class is responsible for producing the profile trees used
throughout the
# rest of this tool. Simply instantiate it and call the parse method
with a
# valid profile string.
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

require 'strscan'

class ProfileParser
  attr_reader :input, :metavariablesymbol, :sourceGroup, :profileTree

  def initialize
    @input = nil
    @metavariablesymbol = Regexp.new("%%")
    @profileTree = []
    @repetition = true
  end

  # Top level parse method. Takes in a profile as a string and returns a
hash with
  # two key/value pairs, "repetition" which indicates whether or not
this profile
  # tree contains a source group repetition construct and "tree" which
is an array
  # that contains hashes which represent constant strings, repetition
constructs,
  # and metavariables. The first part af this method determines if the
profile
  # specifies an alternate metavariable symbol.
  #
  def parse(profileString)
    @input = StringScanner.new(profileString)

    if @input.scan(/METAVARIABLESYMBOL/) != nil
      @input.scan_until(/=/)
      @metavariablesymbol = Regexp.new(@input.scan_until(/\n/).chomp)
    end

    @profileTree = parseSequence()
    return {"repetition" => @repetition, "tree" => @profileTree}
  end

  # Called by parse. Generates the profileTree array.
  #
  def parseSequence
    result = []
```

```
    while(!@input.eos?)
      match = nil
      if (match = @input.scan_until(@metavariablesymbol)) != nil
        result << {"string" => match.chomp("%%")}
        if @input.scan(/\^\d+/) != nil
          result << {"metavariable" => @input.matched.gsub(/\^/,
"").to_i, "new" => true}
        elsif @input.scan(/\d+/) != nil
          result << {"metavariable" => @input.matched.to_i, "new" =>
false}
        elsif @input.scan(/\{/) != nil
          result << {"sourcegroup" => parseLimitedSequence}
          @repetition = false
        elsif @input.scan(/\[/) != nil
          result << parseIntratupleConstruct
        else
          raise "Unrecognized metavariable construct."
        end
      else
        result << {"string" => @input.rest}
        @input.terminate
      end
    end

    return result
  end

  # Returns an array representing the sequence of strings and
metavariable
  # constucts bounded by a repetition construct. Note that source group
repetition
  # is not allowed to be nested within other repetition constucts. Other
then this
  # method is quite similar to parseSequence.
  #
  def parseLimitedSequence
    result = []
    while (match = @input.scan_until(/%%/)) != nil
      result << {"string" => match.chomp("%%")}
      if @input.scan(/\^\d+/) != nil
        result << {"metavariable" => @input.matched.gsub(/\^/, "").to_i,
"new" => true}
      elsif @input.scan(/\d+/) != nil
        result << {"metavariable" => @input.matched.to_i, "new" =>
false}
      elsif @input.scan(/\{/) != nil
        raise "Source group repetition may not be nested within other
repetition constructs."
      elsif @input.scan(/\[/) != nil
        result << parseIntratupleConstruct
      elsif @input.scan(/\}/) != nil
        return result
      else
        raise "Unrecognized metavariable construct."
      end
```

```
      match = nil
    end

    raise "Unbounded repetition construct."
  end

  # Generates a profile tree element representing an intratuple
repetition
  # construct and returns it as a hash.
  #
  def parseIntratupleConstruct
    result = {}
    if (low = @input.scan(/\d+/)) != nil
      if @input.scan(/\.\./) != nil
        if (high = @input.scan(/\d+/)) != nil
          result["low"] = low.to_i
          result["high"] = high.to_i
        else
          raise "Intratuple construct requires a natural number follow
the double dot."
        end
      else
        result['low'] = low.to_i
      end
    else
      raise "Intratuple construct requires a natural number follow the
opening bracket."
    end

    if @input.scan(/\]/) == nil
      raise "Intratuple construct requires a closing bracket follow the
range."
    elsif @input.scan(/\{/) == nil
      raise "Intratuple construct requires an opening brace follow the
range."
    end

    result['intratuple'] = parseLimitedSequence

    return result
  end

end
```

E. DRIVERGENERATOR.RB

```
# This class serves to tie all of the others together. When it is
instantiated
# it takes the tuple string generated by a CTS-C++ program, the profile
tree
# generated by the profileParser class, and a reference to an instance
of the
# chooser class that has been populated by a valid input specification
parse
# tree. The only function that should be called by the user is the
# generateDrivers function which will output one or multiple drivers
into a
# directory called gen_drivers.
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

require 'strscan'

class DriverGenerator
  attr_reader :tupleArray, :profileTree, :chooser, :singleDriver

  def initialize(tupleString, profileTree, chooser)
    @tupleArray = parseTupleString(tupleString)
    @profileTree = profileTree["tree"]
    @multipleDrivers = profileTree["repetition"]
    @chooser = chooser
  end

  # Takes the output of a CTS program as a string and parses the tuples
into
  # an array of arrays. Each sub-array represents a single tuple, and
the main
  # array represents the entire tuple set. Returns the main array. It is
important
  # to note that this method only works on tuples of integers.
  #
  def parseTupleString(tupleString)
    result = []

    tupleString.gsub(tupleString.slice(/.*\n\n/), '').each { |line|
      scanner = StringScanner.new(line)
      tuple = []
      scanner.scan(/\s*/)
      while(!scanner.eos?)
        if scanner.scan(/-1|\d+/) != nil
          tuple << scanner.matched.to_i
        else
          raise("Something is terribly wrong with the tuple string")
        end
```

```
        scanner.scan(/\s*/)
      end

      if tuple.length != 0
        result << tuple
      end
    }

    return result
  end

  # This is the only method that should be called on an instance of this
class.
  # It does a simple check to see if it needs to generate multiple
drivers or a
  # single driver then calls the appropriate method.
  #
  def generateDrivers
    if @multipleDrivers
      generateMultipleDrivers
    else
      generateSingleDriver
    end
  end

  # This method is very similar to the generate code method defined
below. The
  # only real difference is the inclution of the logic for dealing with
source
  # group repetition constructs. Also, this method outputs it's results
directly
  # to a file named "gen_driver" instead of relying on the caller. It
might be
  # possible to roll generateCode and this method into a single method,
and
  # simplify generateSingleDriver to something closer to
generateMultipleDriver.
  #
  def generateSingleDriver
    result = ""
    @profileTree.each { |leaf|
      if leaf["string"] != nil
        result += leaf["string"]
      elsif leaf["metavariable"] != nil
        if leaf["new"] == true
          result += @chooser.newValueFrom(leaf["metavariable"]-1,
tuple[leaf["metavariable"]-1])
        elsif leaf["new"] == false
          result += @chooser.valueFrom(leaf["metavariable"]-1,
tuple[leaf["metavariable"]-1])
        else
          raise "Metavariable construct found without 'new' key." +
leaf.to_s
        end
      elsif leaf["sourcegroup"] != nil
```

```
        @tupleArray.each { |tuple|
          result += generateCode(leaf["sourcegroup"], tuple)
          @chooser.reset
        }
      elsif leaf["intratuple"] != nil
        case leaf.size
        when 2
          (leaf["low"]).times { |n|
            result += generateCode(leaf["intratuple"], tuple)
          }
        when 3
          (rand(leaf["high"] - leaf["low"]) + leaf["low"]).times { |n|
            result += generateCode(leaf["intratuple"], tuple)
          }
        else
          raise "Encountered malformed intratuple repetition construct."
        end
      end
    }

    File.open("./gen_drivers/gen_driver0", "w") { |fd|
      fd << result << "\n"
    }
  end

  # This method constructs a driver string for each tuple in the tuple
array
  # and outputs those strings to files named gen_driver0, gen_driver1,
etc..
  #
  def generateMultipleDrivers
    drivers = []
    @tupleArray.each { |tuple|
      drivers << generateCode(@profileTree, tuple)
      @chooser.reset
    }
    drivers.size.times { |n|
      File.open("./gen_drivers/gen_driver#{n}", "w") { |fd|
        fd << drivers[n] << "\n"
      }
    }
  end

  # This method is used when generating code for multiple drivers. It
takes in
  # a tree in the form of an array of hashes and an array of tuples
(which are
  # themselves arrays). Code is generated for each leaf of the tree in
succession
  # and appended to a result string. If a leaf representing a source
group
  # repetition construct is found an error will be raised because source
group
  # repetition is not allowed when generating multiple drivers. Returns
the result
```

```
  # string after all leaves have been dealt with.
  #
  def generateCode(tree, tuple)
    result = ""
    tree.each { |leaf|
      if leaf["string"] != nil
        result += leaf["string"]
      elsif leaf["metavariable"] != nil
        if leaf["new"] == true
          result += @chooser.newValueFrom(leaf["metavariable"]-1,
tuple[leaf["metavariable"]-1])
        elsif leaf["new"] == false
          result += @chooser.valueFrom(leaf["metavariable"]-1,
tuple[leaf["metavariable"]-1])
        else
          raise "Metavariable construct found without 'new' key." +
leaf.to_s
        end
      elsif leaf["sourcegroup"] != nil
        # Execution should never reach this point under the current
convention
        # of not allowing source group repetition constructs when
generating
        # multiple drivers.
        raise "Source group repetition construct found when generating
multiple drivers."
      elsif leaf["intratuple"] != nil
        case leaf.size
        when 2
          (leaf["low"]).times { |n|
            result += generateCode(leaf["intratuple"], tuple)
          }
        when 3
          (rand(leaf["high"] - leaf["low"]) + leaf["low"]).times { |n|
            result += generateCode(leaf["intratuple"], tuple)
          }
        else
          raise "Encountered malformed intratuple repetition construct."
        end
      end
    }

    return result
  end

end
```

F. DISCRETESET.RB

```
# This class implements the random choice logic for integers and
characters. It
# is only called by the chooser class and should not be independantly
# instantiated. In the future this class may be obsoleted.
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

class DiscreteSet
  attr_reader :values, :register

  def initialize(arrayOfValues)
    @values = arrayOfValues
    @register = nil
  end

  def reset
    @register = nil
  end

  def chooseValue
    if @register == nil
      @register = @values[rand(@values.size)].to_s
      return @register
    else
      return @register
    end
  end

  def chooseNewValue
    return @values[rand(@values.size)].to_s
  end

end
```

G. INFINITESET.RB

```
# It is possible that the implemetation of InfiniteSet is general enough
to be
# used in place of DiscreteSet, thus obsoleting it. If this is the case,
some
# minor changes are needed to the /int/ and /char/ cases of the
Chooser.populate
# method. As it stands this class implements the random choice logic for
the
# float, and double types.
#
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

class InfiniteSet

  def initialize(values)
    @values = values
    @register = nil
  end

  def reset
    @register = nil
  end

  def chooseValue
    if @registor == nil
      case @values.class.to_s
      when "Range"
        @register = (@values.first + random(@values.last -
@values.first)).to_s
      when "Array"
        @register = @values[rand(@values.size)].to_s
      else
        @register = @values.to_s
      end
    else
      return @register
    end
  end

  def chooseNewValue
    case @values.class.to_s
    when "Range"
      @register = (@values.first + random(@values.last -
@values.first)).to_s
    when "Array"
      @register = @values[rand(@values.size)].to_s
    else
```

```
      @register = @values.to_s
    end
  end

  def random(max)
    if max.class.to_s == "Float"
      temp = rand(max) + rand
      if temp > max
        return max
      else
        return temp
      end
    else
      return rand(max+1)
    end
  end

end
```

H. STRINGSET.RB

```
# This class is currently exactly the same as the InfiniteSet class.
# This being the case, ranges for strings will not work, and will in
# fact throw an error. However, lists and singletons should work just
# fine. The reason for this classes existence is to be a place for
# further differentiation of the random choice logic for stirngs. At
# some future date a sane logic for string ranges my be desired, and
# this will be where that logic shall be implemented.
#
#
# Author:: Christopher 'Topher' Eatinger
# Time:: June 2007
# Place:: Naval Postgraduate School
# Version:: 0.1
#

class StringSet

  def initialize(values)
    @values = values
    @register = nil
  end

  def reset
    @register = nil
  end

  def chooseValue
    if @registor == nil
      case @values.class.to_s
      when "Range"
        @register = (@values.first + random(@values.last -
@values.first)).to_s
      when "Array"
        @register = @values[rand(@values.size)].to_s
      else
        @register = @values.to_s
      end
    else
      return @register
    end
  end

  def chooseNewValue
    case @values.class.to_s
    when "Range"
      @register = (@values.first + random(@values.last -
@values.first)).to_s
    when "Array"
      @register = @values[rand(@values.size)].to_s
    else
      @register = @values.to_s
    end
```

```
  end

  def random(max)
    if max.class.to_s == "Float"
      temp = rand(max) + rand
      if temp > max
        return max
      else
        return temp
      end
    else
      return rand(max+1)
    end
  end

end
class StringSet

  def initialize

  end

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Computer Emergency Response Team. “CERT Statistics”. <http://www.cert.org/stats>. 30 April 2007. Last visited: April 2007.

[2] Miller, B. P., Fredriksen, L., So, B. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, Vol. 33, Issue 12, December 1990.

[3] Miller, B. P., Koski, D., et al. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[4] Forrester, J. E., Miller, B. P. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. 4th USENIX Windows Systems Symposium, Seattle, August 2000.

[5] Miller, B. P., Cooksey, G., Moore, F. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. First International Workshop on Random Testing, Portland, Maine, July 2006.

[6] Cohen, D. M., Dalal, S. R., et al. The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, Vol. 13, Issue 5, September 1996.

[7] Hartman, A. Software and Hardware Testing Using Combinatorial Covering Suits. IBM Haifa Research Laboratory, 2003.

[8] Hartman, A., Raskin, L. Problems and Algorithms for Covering Arrays. IBM Haifa Research Laboratory, 2002.

[9] Computer Emergency Response Team. “2006 eCrime Watch Survey”. <http://www.cert.org/archive/pdf/ecrimesurvey06.pdf>. 6 September 2006. Last visited: April 2007.

[10] IBM alphaWorks. “Combinatorial Test Services” <http://www.alphaworks.ibm.com/tech/cts>. 12 January 2004. Last visited: May 2007.

[11] Common Criteria for Information Technology Security Evaluation. Version 3.1, CCIMB-2006-09-[001, 002, 003]. Common Criteria Project Sponsoring Organizations, September 2006.

[12] Nguyen, T. D., Levin, T. E., and Irvine, C. E. TCX Project: High Assurance for Secure Embedded Systems. 11th IEEE Real-Time and Embedded Technology and Applications Symposium Work-In-Progress Session, San Francisco, CA, March 2005.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

4. Chris Eagle
   Naval Postgraduate School
   Monterey, California

5. Dr. Cynthia Irvine
   Naval Postgraduate School
   Monterey, California

6. Christopher Eatinger
   Naval Postgraduate School
   Monterey, California